Прежде чем двигаться дальше, нам нужно глубже познакомиться с механизмом вызова функций, заглянуть за кулису этого процесса, раскрыть некоторые важные детали его работы.

Когда запускается какая-либо программа, то происходит загрузка данных и программного кода в память устройства и, кроме того, автоматически выделяется область памяти под **стек вызова функций**. Его еще называют **стековым фреймом**. Я напомню, что стек – это такая структура, когда новые данные помещаются наверх и извлекаются тоже сверху. Это вроде того, как мы в стакан кладем некие предметы, а затем, вынимаем их тоже сверху. Получаем очередь, работающую по принципу:

LIFO (Last In, First Out) – последний вошел, первый вышел.

Зачем в программе понадобился такой стек? Давайте предположим, что объявлены две функции: max2() – для поиска максимального из двух значений; main() – основная функция, точка входа в программу:

```language-cpp
int max2(int a, int b)
{
	return (a > b) ? a : b;
}

int main(void)
{
	int x = 1, y = 2;
	int res = max2(x, y);

	return 0;
}
```

Каждая функция имеет свои собственные переменные, необходимые для ее работы. Например, в функции main() – это переменные x, y, res, а в функции max2() – это параметры a и b. Так вот, **переменные, связанные с той или иной функцией автоматически создаются в момент вызова этой функции и становятся недоступными после ее завершения**. А **размещаются переменные, как раз в стековом фрейме**. В нашем примере сначала будет вызвана функция main(). В стеке появится блок данных для этой функции. Затем, функция main() вызывает функцию max2() и в стеке появляется еще один блок данных для работы этой второй функции. После завершения функции max2() блок с данными для нее в стековом фрейме перестает быть

актуальным и более не учитывается. Соответственно, параметры a и b становятся недоступными после завершения этой функции. То же самое происходит при завершении функции main(). Все ее данные в стековом фрейме как бы перестают существовать, в том числе и локальные переменные x, y, res.

Получается, что обычные переменные, объявленные внутри функции, автоматически создаются в момент ее вызова и пропадают после ее завершения. Именно поэтому такие переменные получили название **автоматические**. А область их видимости (то есть, существования) ограничивается телом функции. Подробнее об областях видимости и классах памяти мы еще будем говорить.

Давайте теперь в деталях посмотрим, как происходит вызов этих функций. Вначале, **когда вызывается функция main(), вызывающая программа копирует в стек аргументы функции (если они есть), резервирует память под возвращаемое значение и сохраняет адрес возврата, то есть, адрес машинной команды, которую нужно будет выполнить после завершения вызова текущей функции**. Указатель стека ESP, как правило, указывает на позицию в стековом фрейме, где располагается этот адрес. Затем, срабатывает команда call с указанием адреса перехода к подпрограмме, то есть, к первой ячейке памяти, где хранятся команды функции main(). Это и есть непосредственно процесс вызова функции. Сама функция в момент вызова резервирует в стеке память для хранения всех локальных переменных, объявленных внутри нее. В нашем примере – это переменные x, y, res. И дополнительно прописывается еще некоторая служебная информация. После этого начинает отрабатывать логика функции main(). В данном случае вызывать следующую функцию max2(x, y). Соответственно, процесс вызова повторяется. Сначала в стек вызывающая функция, то есть main() добавляет параметры a, b, причем в обратном порядке: сначала b, затем a. Записывает адрес возврата из функции max2() и **перемещает указатель стека ESP на этот новый адрес**. Далее, срабатывает все та же команда call с адресом подпрограммы функции max2(). Функция начинает свою работу и первым делом размещает в стеке свои локальные переменные. Но их нет, поэтому там появляется только служебная информация. После выполнения всех команд подпрограмма функции max2() **доходит до команды ret с переходом на сохраненный адрес возврата**. Мы снова попадаем в функцию main() и ее выполнение продолжается. В частности, восстанавливается прежнее значение указателя стека ESP. В результате, все данные, относящиеся к функции max2(), становятся неактуальными и эта область памяти может быть использована при вызове

других функций. Поэтому продолжать работу с локальными переменными a и b уже нельзя. Собственно, на уровне языка Си они недоступны за пределами тела функции именно по этой причине – они, как бы перестают существовать. Функция main() также завершается, выполняется команда ret с переходом по сохраненному адресу, и мы попадаем в начальный программный блок. Здесь указатель ESP устанавливается на самый низ фреймового стека и программа завершается.

Вот такие манипуляции происходят каждый раз при вызове функций. И мы теперь знаем, что все **обычные локальные переменные и параметры размещаются в стековом фрейме при очередном вызове функции**. При этом память под переменные просто резервируется и не более того. Это значит, что локальные переменные могут принимать произвольные начальные значения, так как в ячейках памяти, которые они занимают, могут находиться любые величины, так называемый шум. Давайте в этом убедимся. Создадим в функции main() целочисленную переменную с именем var и выведем ее на экран:

```cpp
int var;
printf("var = %d\n", var);
```

В моем случае получилось значение:

```cpp
var = 4199120
```

У вас будет какое-то другое. То есть, начальные значения в локальных автоматических переменных непредсказуемы. Это нужно учитывать при составлении программы.

Другой важный вывод связан с тем, что стековый фрейм, как правило, имеет весьма ограниченный размер (несколько мегабайт). Это, во-первых, означает, что мы не можем вызывать бесконечно длинную цепочку функций (одну из другой), так как стек просто заполнится, и при очередном вызове получим ошибку:

Stack Overflow (переполнение стека)

И, во-вторых, не следует внутри функций определять автоматические локальные переменные, которые занимают большой объем памяти. Например, массивы из большого числа элементов. Допустим, такой:

```cpp
int main(void)
{
```

```
        double big_ar[1000000];
        return 0;
}
```

В этом случае памяти стекового фрейма просто не хватит для его хранения и программа завершится аварийно. А вот если объявить вне функций, то никаких проблем не будет.

Забегая вперед отмечу, что данные больших размеров, которые нужно динамически создавать в момент вызова функций, **лучше размещать в основной области памяти, так называемой, куче**. Делается это с помощью вызова функций malloc() и free(), о которых мы с вами еще будем говорить.

На этом мы завершим знакомство с механизмом вызова функций. Думаю, вы теперь хорошо представляете себе особенности и ограничения, связанные с локальными переменными и параметрами функций.